**Transaction Isolation Levels**

**Read Uncommitted**

The transaction can read uncommitted data (i.e., data changed by a different transaction that is still in progress). Dirty reads, nonrepeatable reads, and phantom reads can occur. Bean methods with this isolation level can read uncommitted changes.

**Read Committed**  
The transaction cannot read uncommitted data; data that is being changed by a different transaction cannot be read. Dirty reads are prevented; nonrepeatable reads and phantom reads can occur. Bean methods with this isolation level cannot read uncommitted data.

**Repeatable Read**  
The transaction cannot change data that is being read by a different transaction. Dirty reads and nonrepeatable reads are prevented; phantom reads can occur. Bean methods with this isolation level have the same restrictions as those in the Read Committed level and can execute only repeatable reads.

**Serializable**  
The transaction has exclusive read and update privileges; different transactions can neither read nor write to the same data. Dirty reads, nonrepeatable reads, and phantom reads are prevented. This isolation level is the most restrictive.

**Transaction** **Demarcation:**

Every transaction has a beginning and an end. Beginning a transaction will allow subsequent operations to become a part of the same transaction until the transaction has completed. Transactions can be completed in one of two ways. They can be committed, causing all of the changes to be persisted to the data store, or rolled back, indicating that the changes should be discarded. The act of causing a transaction to either begin or complete is termed transaction demarcation. This is a critical part of writing enterprise applications, because doing transaction demarcation incorrectly is one of the most common sources of performance degradation.

Resource-local transactions are always demarcated explicitly by the application, whereas container transactions can either be demarcated automatically by the container or by using a JTA interface that supports application-controlled demarcation. In the first case, when the container takes over the responsibility of transaction demarcation, we call it container-managed transaction management, but when the application is responsible for demarcation, we call it bean-managed transaction management.

**Transaction Attributes:**

The defined transaction attributes choices are as follows:

* MANDATORY: If this attribute is specified for a method, a transaction is expected to have already been started and be active when the method is called. If no transaction is active, an exception is thrown. This attribute is seldom used, but can be a development tool to catch transaction demarcation errors when it is expected that a transaction should already have been started.
* REQUIRED: This attribute is the most common case in which a method is expected to be in a transaction. The container provides a guarantee that a transaction is active for the method. If one is already active, it is used; if one does not exist, a new transaction is created for the method execution.
* REQUIRES\_NEW: This attribute is used when the method always needs to be in its own transaction; that is, the method should be committed or rolled back independently of methods further up the call stack. It should be used with caution because it can lead to excessive transaction overhead.
* SUPPORTS: Methods marked with supports are not dependent on a transaction, but will tolerate running inside one if it exists. This is an indicator that no transactional resources are accessed in the method.
* NOT\_SUPPORTED: A method marked to not support transactions will cause the container to suspend the current transaction if one is active when the method is called. It implies that the method does not perform transactional operations, but might fail in other ways that could undesirably affect the outcome of a transaction. This is not a commonly used attribute.
* NEVER: A method marked to never support transactions will cause the container to throw an exception if a transaction is active when the method is called. This attribute is very seldom used, but can be a development tool to catch transaction demarcation errors when it is expected that transactions should already have been completed.

Transaction attributes determine exactly what the container-managed transactional behaviour is. Any time the container starts a transaction for a method, the container is assumed to also attempt to commit the transaction at the end of the method. Each time the current transaction must be suspended, the container is responsible for resuming the suspended transaction at the conclusion of the method.

**Persistence Contexts:**

A persistence unit is a named configuration of entity classes. A persistence context is a managed set of entity instances. Every persistence context is associated with a persistence unit, restricting the classes of the managed instances to the set defined by the persistence unit. Saying that an entity instance is managed means that it is contained within a persistence context and it can be acted upon by an entity manager. It is for this reason that we say that an entity manager manages a persistence context.

Understanding the persistence context is the key to understanding the entity manager. An entity’s inclusion or exclusion from a persistence context will determine the outcome of any persistent operations on it. If the persistence context participates in a transaction, the in-memory state of the managed entities will get synchronized to the database. Yet despite the important role that it plays, the persistence context is never actually visible to the application. It is always accessed indirectly through the entity manager and assumed to be there when we need it.

**Entity Managers:**

JPA defines no fewer than three different types of entity managers, each of which has a different approach to persistence context management that is tailored to a different application need. As we will see, the persistence context is just one part of the puzzle.

* **Container-Managed Entity Managers:**

In the Java EE environment, the most common way to acquire an entity manager is by using the @PersistenceContext annotation to inject one. An entity manager obtained in this way is called container-managed because the container manages the lifecycle of the entity manager, typically by proxying the one that it gets from the persistence provider. The application does not have to create it or close it.

Container-managed entity managers come in two varieties. The style of a container-managed entity manager determines how it works with persistence contexts. The first and most common style is called transaction-scoped. This means that the persistence contexts managed by the entity manager are scoped by the active JTA transaction, ending when the transaction is complete. The second style is called extended. Extended entity managers work with a single persistence context that is tied to the lifecycle of a stateful session bean and are scoped to the life of that stateful session bean, potentially spanning multiple transactions.

* **Application-Managed Entity Managers:**

The application, rather than the container, manages the lifecycle of the entity manager. Note that all open entity managers, whether container-managed or application-managed, are associated with an EntityManagerFactory instance. The factory used to create the entity managercan be accessed from the getEntityManagerFactory() call on the EntityManager interface.

Although we expect the majority of applications to be written using container-managed entity managers, application-managed entity managers still have a role to play. They are the only entity manager type available in Java SE, and as we will see, they can be used in Java EE as well. Creating an application-managed entity manager is simple enough. All you need is an EntityManagerFactory to create the instance. What separates Java SE and Java EE for application managed entity managers is not how you create the entity manager but how you get the factory.

**Transaction Management:**

Developing a persistence application is as much about transaction management as it is about objectrelational mapping. Transactions define when new, changed, or removed entities are synchronized to the database. Understanding how persistence contexts interact with transactions is a fundamental part of working with JPA.

Note that we said persistence contexts, not entity managers. There are several different entity manager types, but all use a persistence context internally. The entity manager type determines the lifetime of a persistence context, but all persistence contexts behave the same way when they are associated with a transaction.

There are two transaction-management types supported by JPA. The first is resource-local transactions, which are the native transactions of the JDBC drivers that are referenced by a persistence unit. The second transaction-management type is JTA transactions, which are the transactions of the Java EE server, supporting multiple participating resources, transaction lifecycle management, and distributed XA transactions.

Container-managed entity managers always use JTA transactions, while application-managed entity managers can use either type. Because JTA is typically not available in Java SE applications, the provider needs to support only resource-local transactions in that environment. The default and preferred transaction type for Java EE applications is JTA. As we will describe in the next section, propagating persistence contexts with JTA transactions is a major benefit to enterprise persistence applications.

**Transaction synchronization:**

Transaction synchronization is the process by which a persistence context is registered with a transaction so that the persistence context can be notified when a transaction commits. The provider uses this notification to ensure that a given persistence context is correctly flushed to the database.

**Transaction association:**

Transaction association is the act of binding a persistence context to a transaction. You can also think of this as the active persistence context within the scope of that transaction.

**Transaction propagation:**

Transaction propagation is the process of sharing a persistence context between multiple container-managed entity managers in a single transaction.

There can be only one persistence context associated with and propagated across a JTA transaction. All container-managed entity managers in the same transaction must share the same propagated persistence context.

**Transaction-Scoped Persistence Contexts:**

As the name suggests, a transaction-scoped persistence context is tied to the lifecycle of the transaction. It is created by the container during a transaction and will be closed when the transaction completes. Transaction-scoped entity managers are responsible for creating transaction-scoped persistence contexts automatically when needed. We say only when needed because transaction scoped persistence context creation is lazy. An entity manager will create a persistence context only when a method is invoked on the entity manager and when there is no persistence context available.

When a method is invoked on the transaction-scoped entity manager, it must first see whether there is a propagated persistence context. If one exists, the entity manager uses this persistence context to carry out the operation. If one does not exist, the entity manager requests a new persistence context from the persistence provider and then marks this new persistence context as the propagated persistence context for the transaction before carrying out the method call. All subsequent transaction-scoped entity manager operations, in this component or any other, will thereafter use this newly created persistence context. This behaviour works independently of whether container managed or bean-managed transaction demarcation has been used.

Propagation of the persistence context simplifies the building of enterprise applications. When an entity is updated by a component inside of a transaction, any subsequent references to the same entity will always correspond to the correct instance, no matter what component obtains the entity reference. Propagating the persistence context gives developers the freedom to build loosely coupled applications knowing that they will always get the right data even though they are not sharing the same entity manager instance.

To demonstrate propagation of a transaction-scoped persistence context, we introduce an audit service bean that stores information about a successfully completed transaction. The logTransaction() method ensures that an employee id is valid by attempting to find the employee using the entity manager.

@Stateless

public class AuditServiceBean implements AuditService {

@PersistenceContext(unitName="EmployeeService")

EntityManager em;

public void logTransaction(int empId, String action) {

// verify employee number is valid

if (em.find(Employee.class, empId) == null) {

throw new IllegalArgumentException("Unknown employee id");

}

LogRecord lr = new LogRecord(empId, action);

em.persist(lr);

}

}

After an employee is created, the logTransaction() method of the AuditService session bean is invoked to record the “created employee” event.

@Stateless

public class EmployeeServiceBean implements EmployeeService {

@PersistenceContext(unitName="EmployeeService")

EntityManager em;

@EJB AuditService audit;

public void createEmployee(Employee emp) {

em.persist(emp);

audit.logTransaction(emp.getId(), "created employee");

}

// ...

}

Even though the newly created Employee is not yet in the database, the audit bean can find the entity and verify that it exists. This works because the two beans are actually sharing the same persistence context. The transaction attribute of the createEmployee() method is REQUIRED by default because no attribute has been explicitly set. The container will guarantee that a transaction is started before the method is invoked. When persist() is called on the entity manager, the container checks to see whether a persistence context is already associated with the transaction. Let’s assume in this case that this was the first entity manager operation in the transaction, so the container creates a new persistence context and marks it as the propagated one.

When the logTransaction() method starts, it issues a find() call on the entity manager from the AuditServiceBean. We are guaranteed to be in a transaction because the transaction attribute is also REQUIRED, and the container-managed transaction from createEmployee() has been extended to this method by the container. When the find() method is invoked, the container again checks for an active persistence context. It finds the one created in the createEmployee() method and uses that persistence context to search for the entity. Because the newly created Employee instance is managed by this persistence context, it is returned successfully.

Now consider the case where logTransaction() has been declared with the REQUIRES\_NEW transaction attribute instead of the default REQUIRED. Before the logTransaction() method call starts, the container will suspend the transaction inherited from createEmployee() and start a new transaction. When the find() method is invoked on the entity manager, it will check the current transaction for an active persistence context only to determine that one does not exist. A new persistence context will be created starting with the find() call, and this persistence context will be the active persistence context for the remainder of the logTransaction() call. Because the transaction started in createEmployee() has not yet committed, the newly created Employee instance is not in the database and therefore is not visible to this new persistence context. The find() method will return null, and the logTransaction() method will throw an exception as a result.

The rule of thumb for persistence context propagation is that the persistence context propagates as the JTA transaction propagates. Therefore, it is important to understand not only when transactions begin and end, but also when a business method expects to inherit the transaction context from another method and when doing so would be incorrect. Having a clear plan for transaction management in your application is key to getting the most out of persistence context propagation.

**Extended Persistence Contexts:**

The lifecycle of an extended persistence context is tied to the stateful session bean to which it is bound. Unlike a transaction-scoped entity manager that creates a new persistence context for each transaction, the extended entity manager of a stateful session bean always uses the same persistence context. The stateful session bean is associated with a single extended persistence context that is created when the bean instance is created and closed when the bean instance is removed. This has implications for both the association and propagation characteristics of the extended persistence context.

Transaction association for extended persistence contexts is eager. In the case of container managed transactions, as soon as a method call starts on the bean, the container automatically associates the persistence context with the transaction. Likewise, in the case of bean-managed transactions; as soon as UserTransaction.begin() is invoked within a bean method, the container intercepts the call and performs the same association.

Because a transaction-scoped entity manager will use an existing persistence context associated with the transaction before it will create a new persistence context, it is possible to share an extended persistence context with other transaction-scoped entity managers. As long as the extended persistence context is propagated before any transaction-scoped entity managers are accessed, the same extended persistence context will be shared by all components.

**Persistence Context Collision:**

We said earlier that only one persistence context could be propagated with a JTA transaction. We also said that the extended persistence context would always try to make itself the active persistence context. This can quickly lead to situations in which the two persistence contexts collide with each other. Consider, for example, that a stateless session bean with a transaction-scoped entity manager creates a new persistence context and then invokes a method on a stateful session bean with an extended persistence context. During the eager association of the extended persistence context, the container will check to see whether there is already an active persistence context. If there is, it must be the same as the extended persistence context that it is trying to associate, or an exception will be thrown. In this example, the stateful session bean will find the transaction-scoped persistence context created by the stateless session bean, and the call into the stateful session bean method will fail. There can be only one active persistence context for a transaction.

While extended persistence context propagation is useful if a stateful session bean with an extended persistence context is the first EJB to be invoked in a call chain, it limits the situations in which other components can call into the stateful session bean if they are also using entity managers. This might or might not be common depending on your application architecture, but it is something to keep in mind when planning dependencies between components.

One way to work around this problem is to change the default transaction attribute for the stateful session bean that uses the extended persistence context. If the default transaction attribute is REQUIRES\_NEW, any active transaction will be suspended before the stateful session bean method starts, allowing it to associate its extended persistence context with the new transaction. This is a good strategy if the stateful session bean calls in to other stateless session beans and needs to propagate the persistence context. Note that excessive use of the REQUIRES\_NEW transaction attribute can lead to application performance problems because many more transactions than normal will be created, and active transactions will be suspended and resumed.

If the stateful session bean is largely self-contained; that is, it does not call other session beans and does not need its persistence context propagated, a default transaction attribute type of NOT\_SUPPORTED can be worth considering. In this case, any active transaction will be suspended before the stateful session bean method starts, but no new transaction will be started. If there are some methods that need to write data to the database, those methods can be overridden to use the REQUIRES\_NEW transaction attribute.

**Application-Managed Persistence Contexts:**

Like container-managed persistence contexts, application-managed persistence contexts can be synchronized with JTA transactions. Synchronizing the persistence context with the transaction means that a flush will occur if the transaction commits, but the persistence context will not be considered associated by any container-managed entity managers. There is no limit to the number of application-managed persistence contexts that can be synchronized with a transaction, but only one container-managed persistence context will ever be associated. This is one of the most important differences between application-managed and container-managed entity managers.

An application-managed entity manager participates in a JTA transaction in one of two ways. If the persistence context is created inside the transaction, the persistence provider will automatically synchronize the persistence context with the transaction. If the persistence context was created earlier (outside of a transaction or in a transaction that has since ended), the persistence context can be manually synchronized with the transaction by calling joinTransaction() on the EntityManager interface. Once synchronized, the persistence context will automatically be flushed when the transaction commits.

**Entity Manager Operations:**

**Persisting an Entity:**

The persist() method of the EntityManager interface accepts a new entity instance and causes it to become managed. If the entity to be persisted is already managed by the persistence context, it is ignored. The contains() operation can be used to check whether an entity is already managed, but it is very rare that this should be required. It should not come as a surprise to the application to find out which entities are managed and which are not. The design of the application dictates when entities become managed.

For an entity to be managed does not mean that it is persisted to the database right away. The actual SQL to create the necessary relational data will not be generated until the persistence context is synchronized with the database, typically only when the transaction commits. However, once a new entity is managed, any changes to that entity can be tracked by the persistence context. Whatever state exists on the entity when the transaction commits is what will be written to the database.

When persist() is invoked outside of a transaction, the behavior depends on the type of entity manager. A transaction-scoped entity manager will throw a TransactionRequiredException because there is no persistence context available in which to make the entity managed. Application-managed and extended entity managers will accept the persist request, causing the entity to become managed, but no immediate action will be taken until a new transaction begins and the persistence context becomes synchronized with the transaction. In effect, this queues up the change to happen at a later time. It is only when the transaction commits that changes will be written out to the database.

The persist() operation is intended for new entities that do not already exist in the database. If the provider immediately determines that it is not true, an EntityExistsException will be thrown. If the provider does not make this determination (because it has deferred the existence check and the insert until flush or commit time), and the primary key is in fact a duplicate, an exception will be thrown when the persistence context is synchronized to the database.

**Finding an Entity:**

The ever-present find() method is the workhorse of the entity manager. Whenever an entity needs to be located by its primary key, find() is usually the best way to go. Not only does it have simple semantics, but most persistence providers will also optimize this operation to use an in-memory cache that minimizes trips to the database.

The find() operation returns a managed entity instance in all cases except when invoked outside of a transaction on a transaction-scoped entity manager. In this case, the entity instance is returned in a detached state. It is not associated with any persistence context.

There exists a special version of find() that can be used in one particular situation. That situation is when a relationship is being created between two entities in a one-to-one or many-to-one relationship in which the target entity already exists and its primary key is well known. Because we are only creating a relationship, it might not be necessary to fully load the target entity to create the foreign key reference to it. Only its primary key is required. The getReference() operation can be used for this purpose.Consider the following example:

Department dept = em.getReference(Department.class, 30);

Employee emp = new Employee();

emp.setId(53);

emp.setName("Peter");

emp.setDepartment(dept);

dept.getEmployees().add(emp);

em.persist(emp);

When the getReference() call is invoked, the provider can return a proxy to the Department entity without actually retrieving it from the database. As long as only its primary key is accessed, Department data does not need to be fetched. Instead, when the Employee is persisted, the primary key value will be used to create the foreign key to the corresponding Department entry. The getReference() call is effectively a performance optimization that removes the need to retrieve the target entity instance.

There are some drawbacks to using getReference() that must be understood. The first is that if a proxy is used, it might throw an EntityNotFoundException exception if it is unable to locate the real entity instance when an attribute other than the primary key is accessed. The assumption with getReference() is that you are sure the entity with the correct primary key exists. If, for some reason, an attribute other than the primary key is accessed, and the entity does not exist, an exception will be thrown. A corollary to this is that the object returned from getReference() might not be safe to use if it is no longer managed. If the provider returns a proxy, it will be dependent on there being an active persistence context to load entity state.

Given the very specific situation in which getReference() can be used, find() should be used in virtually all cases. The in-memory cache of a good persistence provider is effective enough that the performance cost of accessing an entity via its primary key will not usually be noticed. In the case of EclipseLink, it has a fully integrated shared object cache, so not only is local persistence context management efficient but also all threads on the same server can benefit from the shared contents of the cache. The getReference() call is a performance optimization that should be used only when there is evidence to suggest that it will actually benefit the application.

**Removing an Entity:**

Employee emp = em.find(Employee.class, empId);

em.remove(emp.getParkingSpace());

When the transaction commits, we see the DELETE statement for the PARKING\_SPACE table get generated, but then we get an exception containing a database error that shows that we have violated a foreign key constraint. It turns out that a referential integrity constraint exists between the EMPLOYEE table and the PARKING\_SPACE table. The row was deleted from the PARKING\_SPACE table, but the corresponding foreign key in the EMPLOYEE table was not set to NULL. To correct the problem we have to explicitly set the parkingSpace attribute of the Employee entity to null before the transaction commits:

Employee emp = em.find(Employee.class, empId);

ParkingSpace ps = emp.getParkingSpace();

emp.setParkingSpace(null);

em.remove(ps);

Relationship maintenance is the responsibility of the application. We will repeat this statement over the course of this book, but it cannot be emphasized enough. Almost every problem related to removing an entity always comes back to this issue. If the entity to be removed is the target of foreign keys in other tables, those foreign keys must be cleared for the remove to succeed. The remove operation will either fail as it did here or it will result in stale data being left in the foreign key columns referring to the removed entity in the event that there is no referential integrity.

An entity can be removed only if it is managed by a persistence context. This means that a transaction-scoped entity manager can be used to remove an entity only if there is an active transaction. Attempting to invoke remove() when there is no transaction will result in a

TransactionRequiredException exception. Like the persist() operation we described earlier, application-managed and extended entity managers can remove an entity outside of a transaction, but the change will not take place in the database until a transaction involving the persistence context is committed.

**Cascading Operations:**

By default, every entity manager operation applies only to the entity supplied as an argument to the operation. The operation will not cascade to other entities that have a relationship with the entity that is being operated on. For some operations, such as remove(), this is usually the desired behavior. We wouldn’t want the entity manager to make incorrect assumptions about which entity instances should be removed as a side effect from some other operation. But the same does not hold true for operations such as persist(). Chances are that if we have a new entity and it has a relationship to another new entity, the two must be persisted together.

Entity manager operations are identified using the CascadeType enumerated type when listed as part of the cascade attribute. The PERSIST, REFRESH, REMOVE, MERGE, and DETACH constants pertain to the entity manager operation of the same name. The constant ALL is shorthand for declaring that all five operations should be cascaded. By default, relationships have an empty cascade set.

**Detachment and Merging:**

Simply put, a detached entity is one that is no longer associated with a persistence context. It was managed at one point, but the persistence context might have ended or the entity might have been transformed so that it has lost its association with the persistence context that used to manage it. The persistence context, if there still is one, is no longer tracking the entity. Any changes made to the entity won’t be persisted to the database, but all the state that was there on the entity when it was detached can still be used by the application.

A detached entity cannot be used with any entity manager operation that requires a managed instance.

The opposite of detachment is merging. Merging is the process by which an entity manager integrates detached entity state into a persistence context. Any changes to entity state that were made on the detached entity overwrite the current values in the persistence context. When the transaction commits, those changes will be persisted. Merging allows entities to be changed “offline” and then have those changes incorporated later on.

The following sections will describe detachment and how detached entities can be merged back into a persistence context.

**Detachment:**

There are two views of detachment. On one hand, it is a powerful tool that can be leveraged by applications in order to work with remote applications or to support access to entity data long after a transaction has ended. On the other hand, it can be a frustrating problem when the domain model contains lots of lazy-loading attributes and clients using the detached entities need to access this information.

There are many ways in which an entity can become detached. Each of the following situations will lead to detached entities:

* When the transaction that a transaction-scoped persistence context is associated with commits, all the entities managed by the persistence context become detached.
* If an application-managed persistence context is closed, all its managed entities become detached.
* If a stateful session bean with an extended persistence context is removed, all its managed entities become detached.
* If the clear() method of an entity manager is used, it detaches all the entities in the persistence context managed by that entity manager.
* If the detach() method of an entity manager is used, it detaches a single entity instance from the persistence context managed by that entity manager.
* When transaction rollback occurs, it causes all entities in all persistence contexts associated with the transaction to become detached.
* When an entity is serialized, the serialized form of the entity is detached from its persistence context.

**Merging Detached Entities:**

The merge() operation is used to merge the state of a detached entity into a persistence context. The method is straightforward to use, requiring only the detached entity instance as an argument. There are some subtleties to using merge() that make it different to use from other entity manager methods. Consider the following example, which shows a session bean method that accepts a detached Employee parameter and merges it into the current persistence context:

public void updateEmployee(Employee emp) {

em.merge(emp);

emp.setLastAccessTime(new Date());

}

Assuming that a transaction begins and ends with this method call, any changes made to the Employee instance while it was detached will be written to the database. What will not be written, however, is the change to the last access time. The argument to merge() does not become managed as a result of the merge. A different managed entity (either a new instance or an existing managed version already in the persistence context) is updated to match the argument, and then this instance is returned from the merge() method. Therefore to capture this change, we need to use the return value from merge() because it is the managed entity. The following example shows the correct implementation:

public void updateEmployee(Employee emp) {

Employee managedEmp = em.merge(emp);

managedEmp.setLastAccessTime(new Date());

}

**Cascading Operations:**

By default, every entity manager operation applies only to the entity supplied as an argument to the operation. The operation will not cascade to other entities that have a relationship with the entity that is being operated on. For some operations, such as remove(), this is usually the desired behavior. We wouldn’t want the entity manager to make incorrect assumptions about which entity instances should be removed as a side effect from some other operation. But the same does not hold true for operations such as persist(). Chances are that if we have a new entity and it has a relationship to another new entity, the two must be persisted together.

Entity manager operations are identified using the CascadeType enumerated type when listed as part of the cascade attribute. The PERSIST, REFRESH, REMOVE, MERGE, and DETACH constants pertain to the entity manager operation of the same name. The constant ALL is shorthand for declaring that all five operations should be cascaded. By default, relationships have an empty cascade set.